Formal Specification Techniques for Object-Oriented Programs

Peter Müller and Arnd Poetzsch-Heffter*

FernUniversität
D-58084 Hagen

Abstract. Specification techniques for object-oriented programs relate the operational world of programs to the declarative world of specifications. We present a formal foundation of interface specification languages. Based on the formal foundation, we develop new specification techniques to describe functional behavior, invariants, and side-effects. Furthermore, we discuss the influence of program extensions on program correctness.

1 Introduction

Interface specification techniques have been developed for the precise documentation of program behavior ([GH93, FZZ96, PH95]) and as a tool for program design ([Jon91]). Interface specifications relate the operational, state-based world of programs to the declarative, state-less world of universal specifications.

Interface specifications state program properties in an abstract, declarative way and allow one to formally prove that programs satisfy these properties. This extended abstract develops a formally-founded interface specification technique for object-oriented programs that can be used for program verification. It fulfills the following requirements entailed by the goal of formal program verification: 1. Specifications must have a formal semantics to enable formal proofs. 2. Specifications of certain program components must enable one to verify programs that make use of these components. I. e., specifications must be detailed enough to describe all important effects, in particular side-effects on the environment. 3. The connection between interface specifications and proof obligations must be clear. 4. Specifications and correctness proofs should stay valid if the underlying program is extended. With subtyping this is in general not the case. In this extended abstract, we especially show how the requirements above influence interface specification techniques. We illustrate the investigation steps by a small example program written in a C++ subset and use the Larch specification language for C++ as a starting point of our analysis (cf. [Lea96]), because this is one of the most advanced interface specification languages. For verification, we assume a Hoare-style logic (cf. [Hoa69]).
The main contributions of this paper are:
1. Improvement of existing specification techniques towards formal verification.
2. Formal semantics of interface specifications, in particular of class invariants.

* [Peter.Mueller, Arnd.Poetzsch-Heffter]@fernuni-hagen.de

3. Short analysis of behavioral subtyping in the context of verification.
4. New techniques to specify sharing and environmental properties of methods.

Related Work A lot of work has been done aiming at the construction of correct software. Some approaches concentrate on top-down software development by iteratively refining specifications until an executable program is reached (cf. the refinement calculus by Back [Bac88], the KorSo project [BJ95]). In contrast, our framework relates universal specifications and programs (implementing abstract data types via concrete and abstract classes) without enforcing a certain style of software development.

Our work has been inspired by Larch (cf. [GH93]). As in our approach, Larch specifications consist of two major parts: (a) A program-independent specification of abstract data types and (b) a program dependent part that relates the implementation to the abstract data types. In the implementation, the ADT values are in general represented by several linked objects. An interface specification of a class C consists of an invariant and specifications for C's methods.

In Larch-style specifications, the *functional behavior* of methods is specified by describing the input/output behavior based on the abstract values represented by the parameter and result objects. This is done by pre- and postconditions, which are first-order formulae. The *environmental behavior* of methods is expressed via so-called *modifies clauses*. These are lists of all objects that may be changed by a method. Properties that must hold for all objects of a type can be specified as class invariants (see section 3.2).

Compared to Larch, our framework has three major advantages: 1. We give specifications a formal semantics, which most Larch specification languages don't. A formal semantics is indispensable for verification. 2. We provide more elaborated techniques for the specification of side-effects and sharing. 3. We use explicit abstraction functions. This aspect is crucial for verification and will be illustrated in the following: Consider the example in appendix A. We have a class `database` that implements an abstract type *Database*[2]. In Larch, the typical way to specify the functional behavior of method `emptyDB`, which returns an empty database, would be as follows (the example is not in Larch/C++ syntax):

```
database *emptyDB()
pre  true
post result = empty
```

The important aspect with the above, almost trivial specification is the implicit *abstraction* that is applied to the result: The result, which is an object of the programming language, is equated with a value of the abstract data type. To verify that `emptyDB` fulfills the specification, we have to make the abstraction explicit using an abstraction function. Abstraction functions map objects of the programming language to the universal specification framework. Section 2 describes the basic techniques to define such functions.

[2] Identifiers of programs are printed in `typewriter` font, whereas names of the abstract type are printed *italic*.

Overview This extended abstract is organized as follows: Section 2 investigates the needed semantical aspects of the underlying programming language. In section 3, we show how method behavior and class invariants can be specified. Furthermore, we present a formal semantics of specifications. Section 4 discusses the effects of program extensions. The conclusions are contained in section 5.

2 Formalizing Environments and Abstraction Functions

This section describes how the data and state model of an object-oriented programming language can be formalized and how abstraction functions can be defined based on such a formal model. We have to focus on the central techniques and ideas. In particular, we cannot go into details about a formalization of C++, but assume only a restricted language where each object can be considered as a pointer to a record. We use many-sorted first-order logic with recursive data types for the specification. For details, we have to refer to [PH97].

Object Environments The data model of a programming language defines the objects and values that may be used in programs. To keep things simple, we consider predefined values like integers or booleans as objects without attributes that exist in initial program states and cannot be created or deleted. We assume a sort *Type* containing a symbol for each type defined in a program and a sort *Object* containing: (a) for each user-defined type an infinite set of objects, (b) for each user-defined type a null object, and (c) the predefined values. The function $typ : \mathit{Object} \to \mathit{Type}$ yields for each object its type symbol; the predicate $isnull : \mathit{Object} \to \mathit{Boolean}$ checks whether an object is a null object. Furthermore, we assume a sort *Location*: A location is a pair (X, A) — denoted by $X.A$ — where X is a user-defined object and A is an attribute of the class of X. Locations are the formal counterparts of instance variables (or data members). The function $obj : \mathit{Location} \to \mathit{Object}$ yields the object a location belongs to ($obj(X.A) = X$).

Objects have states. The state of an object tells whether the object is alive or not yet allocated, and it assigns an object to each of its locations. The collection of all object states at a point of program execution is called the current *object environment*. Object environments are modelled via the abstract data type *ObjEnv*. The following operations are defined on object environments: $E\langle L := X\rangle$ denotes the environment after updating environment E at location L with object X. $E(L)$ denotes the object read from location L in environment E. $new(E, T)$ returns a new object of type T in environment E. $E\langle T\rangle$ denotes the environment after allocating a new object of type T in environment E. $alive(X, E)$ checks whether object X is alive in environment E. [PH97] presents an axiomatization of these operations.

Predicates on Object Environments The update of a location L affects properties of all objects that reference L: E. g., modifying a location of a list element X in a singly linked list affects all lists for which X is an element. On the other hand, if only locations are modified that are not reached by an object X, we

know that the properties of X remain invariant under these modifications. Consequently, reachability is a central property for verification. It is formalized as a predicate expressing that an object X reaches a location L in an environment E: $reach(X, L, E) \Leftrightarrow_{def} obj(L) = X \vee \exists K : obj(K) = X \wedge reach(E(K), L, E)$. Based on *reach*, we can define a predicate *disj* expressing that the set of objects reachable from object X is disjoint from those objects reachable from Y: $disj(X, Y, E) \Leftrightarrow_{def} \forall L : \neg reach(X, L, E) \vee \neg reach(Y, L, E)$. Beside being indispensable for verification, the formal specification of object environments allows to use the vocabulary provided by the abstract data type *ObjEnv* in interface specifications. Thus, interface specifications become more flexible and can support different levels of abstraction down to the lowest level of abstraction, namely the object level. An example illustrating this feature can be found in section 3.

Abstraction functions An abstraction function maps an object X in an environment E to the abstract value that is represented by X (and possibly some other objects reachable from X) in E. Based on the formalization of object environments, abstraction functions can be defined in a precise way. E. g., the abstraction function *aDB* maps objects of class `database` (see appendix) to values of sort *Database* (*empty* and *insert* are the constructors of *Database*, see appendix). *aDB* is specified as binary function $aDB : Object \times ObjEnv \to Database$:

$$
\begin{array}{l}
typ(X) \preceq \texttt{database} \wedge E(X.length) = 0 \Rightarrow \ \ aDB(X, E) = empty \\
typ(X) \preceq \texttt{database} \wedge E(X.length) > 0 \Rightarrow \\
\quad aDB(X, E) = insert(\ aDB(E(X.link), E),\ aDATA(E(X.elem), E)\ )
\end{array}
$$

where $\preceq$ denotes the subtype relation on the types in a program and *aDATA* is the abstraction for objects of type `data`.

3 Interface Specifications

The first part of this section focuses on the specification of method behavior. In the second subsection, we show how well-formedness of data representations can be expressed by class invariants. After that, we present a formal meaning of specifications by interpreting them as proof obligations in a Hoare-style logic.

3.1 Specifying Method Properties

Method behavior has three different aspects: (a) functional behavior, i. e., the relation between the abstract values represented by the parameters in the prestate and the abstract value of the result in the poststate; (b) environmental properties expressing which parts of the environment change under method execution; (c) sharing properties relating the representations of parameters and result.

Specification of Functional Behavior To refer to the object environment in pre- and postconditions, we use the symbol \$ of sort *ObjEnv*; \$ can be considered as a global variable and has usually different values in pre- and poststates (one can think of \$ representing the object store). Using this notation and the abstraction function *aDB*, the intention of the specification for method `emptyDB` given in section 1 can be made explicit. Trivial preconditions (identical to *true*) will be omitted in the following:

`database *emptyDB()`
`post` $aDB(\text{result}, \$) = empty$

The typical specification of functional method behavior expresses the abstraction of the result as a term over the abstractions of the parameters in the prestate. The prestate values of parameters and of the environment variable can be used in postconditions by using a prestate-operator, denoted by "^". We illustrate this by the specification of method `insertDB`:

`database *insertDB(data *d)`
`post` $aDB(\text{result}, \$) = insert(\, aDB(\text{this}\hat{}, \$\hat{}),\ aDATA(\text{d}\hat{}, \$\hat{})\,)$

Specification of Environmental Behavior In Larch/C++, environmental properties are expressed by modifies clauses. A modifies clause lists all objects which may be changed under execution of a method by enumeration or by the reach(X) construct, which denotes all objects reachable from X. The disadvantage of this technique is that sharing is not taken into account. Consider a method that updates the last element of a singly linked list `l`. In fact, it modifies all lists referencing the last element, which are at least as many as the length of `l`. This property is very difficult to express by modifies clauses.

In our framework, the explicit object environment can be used to specify environmental properties. E. g., the following specification of `emptyDB` precisely describes the side-effects, namely the creation of a new `database` object:

`database *emptyDB()`
`post` $\text{result} = new(\$\hat{}, \texttt{database}) \ \wedge\ \$ = \$\hat{}\langle\texttt{database}\rangle$

The absence of any side-effects can be specified by conjoining $\$ = \$\hat{}$ to the postcondition. This means that neither any locations nor liveness of any objects are changed. A more interesting, typical environmental property is that a method only modifies objects of the class it belongs to. For the methods of class `database`, this can be expressed by conjoining $typ(obj(L)) \neq \texttt{database} \Rightarrow \$(L) = \$\hat{}(L)$ to the postconditions. More advanced techniques for specification of environmental properties are presented in [PH97].

Specification of Sharing Properties Many realistic implementations use so-called *destructive updates* of data representations to increase efficiency. E. g., class `database` provides a method `updateDB` manipulating one entry. Such an update affects the abstract values represented by all objects referencing the updated entry. As a counterpart to destructive updates, we usually find methods to clone

or copy whole object structures. We use the predicate *disj* to specify that `copyDB` creates a completely new object structure representing the same abstract value:

`database* copyDB()`
`post` $aDB(\text{result}, \$) = aDB(\text{this}\hat{}, \$\hat{}) \ \wedge \ disj(\text{result}, \text{this}\hat{}, \$)$

Such properties cannot be expressed in many specification frameworks as they presuppose the distinction between the abstract and the representation level (for a more detailed treatment of sharing properties, we refer to [PH97]).

3.2 Class Invariants

Abstraction of object structures only works if the object structures are *well-formed*. E. g., abstraction of `database` objects is only defined if they are not null and if the linked object list is acyclic. Well-formedness of `database` objects can be defined as follows (*wfDATA* expresses the well-formedness of data elements):

$$\begin{aligned} wfDB(X,E) \ \Leftrightarrow_{def} \ & \neg isnull(X) \ \wedge \ typ(X) \preceq \texttt{database} \ \wedge \\ & \big(\ E(X.length) = 0 \ \vee \ (\ \ E(X.length) > 0 \ \wedge \ wfDATA(E(X.elem)) \wedge \\ & wfDB(E(X.link)) \ \wedge \ E(X.length) = E(\,E(X.link).length\,) + 1 \ \ ) \quad \big) \end{aligned}$$

Well-formedness is a typical invariance property, i. e., a property that has to hold for all objects of a class. Thus, we use $wfDB(X,E)$ as *class invariant* of `database`; i. e., the invariant is a binary predicate $inv : Object \times ObjEnv \rightarrow Boolean$. The meaning of such invariants is discussed and explained in the next subsection.

3.3 Meaning of Interface Specifications

The meaning of invariants can be made precise by answering three questions: 1. For which objects must the invariants hold? 2. In which execution states must the invariants hold? 3. Which invariant has to hold for which method? The invariant inv_C of a class C has to hold for all non-null, living objects of type C; we abbreviate this by predicate INV_C:

$$INV_C(E) \Leftrightarrow_{def} \forall X : typ(X) \preceq C \wedge alive(X,E) \wedge \neg isnull(X) \Rightarrow inv_C(X,E)$$

Concerning the second question, an invariant of class C certainly need not hold in all intermediate states during execution of C's methods; in particular during the construction of linked object structures, invariants are usually violated. But we expect them to express properties that are invariant under method execution: I. e., if the invariants hold for all objects in the precondition of a *public* method, they should hold in the postcondition. In particular, they have to hold for objects created during method execution. We require invariance only for public methods, because this guarantees that the invariant of a class C holds outside the execution of methods of C and because we want to allow private methods to perform auxiliary operations violating e. g. well-formedness properties.

From a verification point of view, the answer to the third question is fairly simple: To use class invariants as invariants in the proof technical sense, they

have (a) to be true in possible initial program states and they have (b) to be invariant under *all* public methods. Requirement (a) is trivially satisfied because no user-defined objects are alive in initial program states. Requirement (b) is the proof obligation resulting from invariants. Although requirement (b) is as well justified from an operational point of view — a method m_C of class C can call a method m_D of class D and thus manipulate D-objects —, the literature often assigns a weaker meaning to invariants which makes verification much more difficult and leads to unintuitive situations.

We define the formal semantics of specifications by interpreting them as triples in a Hoare-style logic which is a formalization of the axiomatic semantics of the underlying programming language. Thus, the connection between specifications and programs is precisely defined and verification can be done by proving the resulting triples in the programming logic. Let us assume a program **P** with classes $C_1 \ldots C_n$. The specification of **P** consists of the class invariants INV_{C_i} and of a pre-postcondition-pair (R_m, Q_m) for each method m. To verify **P**, we have to prove a triple of the following form for each public method m.

$$\{\, R_m \wedge \bigwedge_{i=1}^{n} INV_{C_i}(\$) \,\} \text{ meth m } \{\, Q_m \wedge \bigwedge_{i=1}^{n} INV_{C_i}(\$) \,\}$$

4 Program Extensions

In this section, we analyze the effects of program extensions on verified programs. We show how correctness can be preserved by behavioral subtyping. Furthermore, we summarize and discuss the proof obligations occurring from program extensions.

Behavioral Subtyping In object-oriented programming languages, correctness of programs can be affected by adding new classes as subtypes of existing classes. This effect is due to dynamic binding: Methods of the new subtype may be called in contexts where initially only methods of existing types could occur. Thus, we enforce subtypes to satisfy the specification of their supertypes. In the literature, this notion is usually called behavioral subtyping (cf. e. g. [LW94]). To keep things simple, we use the following definition: Type S is a behavioral subtype of T if (1) $inv_S(X, E)$ implies $inv_T(X, E)$ for all X of type S and if (2) for each method m associated with S and T with pre-post-pairs (R_S, Q_S) and (R_T, Q_T), R_T implies R_S and Q_S implies Q_T in case the `this` object is of type S. I. e., $S :: m$ shows the behavior specified for $T :: m$. Classes that are derived from superclasses for the reason of "pure" subtyping (i. e. not for the reason of code inheritance) are usually intended to be behavioral subtypes.

We illustrate behavioral subtyping by extending our database example. Class `rdatabase` is a subclass and thus a subtype of class `database`. It extends `database` by storing for each element the number of accesses.

```
class rdatabase : public database {
  protected:  int        access_count;
```

```
public:     rdatabase *insertDB(data *d);
            int        freqDB(int i);        };
```

For brevity, the signatures of the constructor and methods `emptyDB` and `accessDB` are omitted. To reflect the additional information of `rdatabase` on the abstract level as well, we assume a corresponding abstract data type *RDatabase* having essentially[3] the same operations as *Database* except that they work on values of sort *RDatabase* and that there is an additional operation to read out the access count. To distinguish the *RDatabase*-operations from those of *Database*, we prefix them with an "r". The abstraction function for `rdatabase` objects has the following signature: $aRDB : Object \times ObjEnv \rightarrow RDatabase$. Thereby, the functional behavior of `rdatabase::insert` can simply be specified as:

`rdatabase *insertDB(data *d)`
`post` $aRDB(\text{result}, \$) = rinsert(\ aRDB(\text{this}\hat{}, \$\hat{}),\ aDATA(\text{d}\hat{}, \$\hat{})\ )$

The class invariant of `rdatabase` is assumed to be the same as that of `database`. Does `rdatabase::insert` obey the rules of behavioral subtyping? The implication is trivially true for the precondition[4]. What remains to be shown is

$aRDB(\text{result}, \$) = rinsert(\ aRDB(\text{this}\hat{}, \$\hat{}),\ aDATA(\text{d}\hat{}, \$\hat{})\ )$
$\Rightarrow \quad aDB(\text{result}, \$) = insert(\ aDB(\text{this}\hat{}, \$\hat{}),\ aDATA(\text{d}\hat{}, \$\hat{})\ )$

To prove the implication, we have to relate terms of sort *RDatabase* to terms of sort *Database*. As it is typical for the relation between super- and subtypes, *RDatabase* is a specialization of *Database*. We assume a mapping $rdbtodb : RDatabase \rightarrow Database$ that forgets the access count information. Functions like *rdbtodb* are often called coercion functions . They relate the abstract level of sub- and supertypes and have to satisfy homomorphism properties. E. g., to prove the implication resulting from the behavioral subtype constraint, the following two properties of *rdbtodb* are needed:

$rdbtodb(\ rinsert(RDB, D)\ ) = insert(\ rdbtodb(RDB), D\ )$
$typ(X) \preceq$ `rdatabase` $\Rightarrow \quad aDB(X, E) = rdbtodb(\ aRDB(X, E)\ )$

Based on these two properties, it is easy to show that `rdatabase::insert` fulfills the constraints of behavioral subtyping. Generally spoken, the rules of behavioral subtyping allow to prove that methods of subtypes behave like the corresponding methods of supertypes. Therefore, correctness of the extended program is not affected as long as all invariants are preserved.

Invariants and Program Extension Beside the above proof obligations for adding subtypes, other obligations concerning class invariants occur whenever a new class is added to a program. Assume, we have a verified program **P** with classes $C_1 \ldots C_n$. We extend this program by a new class C which is not necessarily a subtype of an existing class. Let *INV* denote the conjunction of the invariants of

[3] Some minor changes have to be done in order to manage the access count.
[4] Recall that omitted preconditions are identical to *true*.

$C_1 \ldots C_n$ and let INV_C denote the invariant of C. As pointed out above, we have to prove that every method preserves every invariant; essentially, this results in the following proof obligations: 1. C_i::m preserves INV. 2. C_i::m preserves INV_C. 3. C::m preserves INV. 4. C::m preserves INV_C. Obligation 1 is already proved as **P** is verified. Obligations 3 and 4 belong to the verification of the new methods C::m. Unpleasantly, obligation 2 may cause to revisit an already proven program. This should be avoided because the implementation of **P** may come from a class library and may not be accessible.

Precise specifications of environmental properties (see section 3) allow to prove that methods preserve invariants of new classes without having to revisit the method implementations themselves.

5 Conclusion

This paper presented formal foundations for interface specifications and illustrated their use for the verification of object-oriented programs. The benefits of formal foundations can be summarized as follows:

- An integrated formal foundation for concrete data representations and abstract specifications is needed to give interface specifications a precise meaning. Abstraction functions are used to relate both worlds.
- Specifications must be able to refer to concrete data representations, e. g., to express well-formedness and to define abstraction functions. Therefore, the data and state model should be accessible within interface specifications.
- Verification requires to specify different aspects of methods, in particular functional behavior and environmental properties.
- The formal semantics of specifications can be described by transforming them into triples of a Hoare-style logic. Verification is done by proving this triples in the logic.
- To preserve correctness of programs under extension, subtypes should be behavioral subtypes. By a sufficiently strong specification of environmental properties, proof obligations coming from program extensions can be discarded without having to revisit already verified implementation parts.

Future work will mainly be concerned with the automation of correctness proofs by using Dijkstra's weakest precondition technique. Furthermore, we aim at the integration of tools for specification and verification into so-called logic-based programming environments.

A Appendix

This appendix presents the following C++ example program implementing a primitive database. The database supports methods to insert, access, and update database entries. It is represented by a singly linked list of its elements. The length of the list is stored explicitly.

```
class database {
  protected:  data     *elem;
              database *link;
              int       length;
  public:     static virtual database *emptyDB();
              virtual database *insertDB( data *d );
              virtual void     *updateDB( int key, data *d);
              virtual database *copyDB();                  };
```

For brevity, we omitted the signatures of the constructor and methods `isemptyDB`, `iselemDB`, and `accessDB`. Class `data` is also not showed here. It provides a method `key` that returns for each `data`-object a unique integer-valued key, which is used to identify elements in the database. To specify the interface of class `database`, we use the abstract data type *Database*. Among others, it contains the following functions:

$$\begin{aligned} &\mathit{empty} : \rightarrow \mathit{Database} \\ &\mathit{insert} : \mathit{Database} \times \mathit{data} \rightarrow \mathit{Database} \\ &\mathit{key} \quad : \mathit{data} \rightarrow \mathit{Integer} \end{aligned}$$

References

[Bac88] R. J. R. Back. A calculus of refinement for program derivations. *Acta Informatica*, 25:593–624, 1988.

[BJ95] Manfed Broy and Stefan Jähnichen, editors. *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[FZZ96] A. Frick, W. Zimmer, and W. Zimmermann. Konstruktion robuster und flexibler Klassenbibliotheken. *Informatik — Forschung und Entwicklung*, 11:168–178, 1996.

[GH93] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[Jon91] H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, LNCS 551, pages 428–456. Springer-Verlag, 1991.

[Lea96] Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Hiam Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996.

[LW94] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[PH95] Arnd Poetzsch-Heffter. Interface specifications for program modules supporting selective updates and sharing and their use in correctness proofs. In G. Snelting, editor, *Softwaretechnik 95*, 1995.

[PH97] Arnd Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. Technische Universität München, 1997. (to appear).